Exploiting Annotations for the Rapid Development of Collaborative Web Applications

Matthias Heinrich
SAP AG
SAP Research Dresden
matthias.heinrich@sap.com

Franz Josef Grüneberger
SAP AG
SAP Research Dresden
franz.josef.grueneberger@sap.com

Thomas Springer
Department of Computer Science
Dresden University of Technology
thomas.springer@tu-dresden.de

Martin Gaedke
Department of Computer Science
Chemnitz University of Technology
martin.gaedke@cs.tu-chemnitz.de

ABSTRACT

Web application frameworks are a proven means to accelerate the development of interactive web applications. However, implementing collaborative real-time applications like Google Docs requires specific concurrency control services (i.e. document synchronization and conflict resolution) that are not included in prevalent general-purpose frameworks like jQuery or Knockout. Hence, developers have to get familiar with specific collaboration frameworks (e.g. ShareJS) which substantially increases the development effort. To ease the development of collaborative web applications, we propose a set of source code annotations representing a lightweight mechanism to introduce concurrency control services into mature web frameworks. Those annotations are interpreted at runtime by a dedicated collaboration engine to sync documents and resolve conflicts. We enhanced the general-purpose framework Knockout with a collaboration engine and conducted a developer study comparing our approach to a traditional concurrency control library. The evaluation results show that the effort to incorporate collaboration capabilities into a web application can be reduced by up to 40 percent using the annotation-based solution.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Computer supported cooperative work, Synchronous interaction, Web-based interaction*

Keywords

Groupware, Shared Editing, Web Applications, Web Engineering

1. INTRODUCTION

Collaborative web applications are pervasive in our daily lives since they exhibit numerous advantages in contrast to traditional desktop applications. Leveraging the web as application platform provides access from a myriad of devices (e.g. PCs, smartphones, etc.) and allows for immediate adoption without requiring time-consuming installation procedures. Moreover, providing real-time collaboration features allowing multiple users to edit the very same document simultaneously supersedes conventional document merging or document locking techniques. These benefits altogether prompted prominent collaborative web applications like Google Docs serving millions of users each and every day.

Even though there is a variety of shared editing use cases (e.g. jointly create documents, spreadsheets, presentations, source code files, CAD models, etc.), web-based collaboration tools offering shared editing capabilities are rare. From the plentitude of web applications – for example the Chrome Web Store lists more than 6 000 – we could only identify very few web applications targeting some form of shared editing. We attribute this mismatch of existing applications to potential use cases to the poor technology support. In particular, web frameworks do not support concurrency control services to the same extent they support common programming tasks like form validation or asynchronous communication. On the one hand, general-purpose frameworks like jQuery or Knockout do not supply any functionality regarding concurrency control; on the other hand, specific collaboration frameworks like ShareJS offer only a very limited set of features (e.g. only strings can be synchronized). Hence, programmers face the dilemma of having to get familiar with an extra framework which might not even suit their needs. In essence, having to learn a specific collaboration framework or implementing the required collaboration capabilities from scratch is a time-consuming and costly endeavor.

To lower the development costs implementing collaboration capabilities, we propose the Collaborative Applications via Data Annotations (ColADA) approach. ColADA leverages source code annotations to enrich existing general-purpose web frameworks with concurrency control support. Annotations are a specific form of metadata and represent a widely adopted, lightweight approach to enhance applications with additional runtime features that are supplied by a dedicated annotation engine. For example, Java frameworks like Hibernate or Spring provide annotations (e.g. @Table, @Column, etc.) accompanied by an annotation engine to enhance applications with features like object persistence, dependency injection or container configuration.

We adopt source code annotations as a means to configure a collaboration engine. While developers enrich the application's source code with annotations to declare data objects as synchronized, the ColADA collaboration engine processes these annotations and accomplishes the synchronization. The implemented ColADA system enriches the widely-adopted Knockout framework with collaboration capabilities. In contrast to using a comprehensive programming library, we assume that a minimal annotation language eases learnability and reduces development time as well as boilerplate code. Moreover, an annotation-based solution allows for multiple product variants. While evaluating annotations activates the collaboration feature, removing annotations results in a fully functional single-user application.

To confirm our assumptions that an annotation-based solution can speed up the development of collaborative web applications, we conducted a thorough developer study involving eight programmers which had to use the ColADA solution and a conventional concurrency control library to develop a collaborative web application.

The main contributions of this paper are threefold:

- We propose an annotation-based approach to enrich existing general-purpose frameworks with concurrency control support to speed up the development of collaborative web applications.
- We report on a developer study with eight programmers comparing the developer productivity of an annotation-based approach with a traditional approach leveraging a specific collaboration library.
- We carve out the framework characteristics that are required to incorporate a collaboration engine for the rapid development of shared editing applications and discuss the entailed benefits as well as the induced limitations.

The rest of this paper is organized as follows: Section 2 exposes related work and Section 3 introduces the design goals of the ColADA system as well as the ColADA architecture. Section 4 elaborates on the ColADA implementation and Section 5 shows the results of the conducted developer study considering development time, lines of code, etc. While Section 6 discusses limitations of the proposed approach and requirements to adopt the ColADA system, Section 7 draws conclusions.

2. RELATED WORK

There are various approaches aiming to accelerate programmer productivity when developing collaborative applications. Those approaches can be divided into two main categories: (1) concurrency control libraries exposing a set of low-level methods and (2) transformation approaches capable of converting single-user applications into collaborative multi-user applications. In this section, we will introduce examples for each category and compare them to our work.

All approaches supporting the implementation of shared editing applications (e.g. collaborative text editors, shared whiteboards, etc.) have to expose document synchronization and conflict resolution services. While the synchronization ensures that the input from various geographically dispersed users is synchronized without notable delay, the conflict resolution allows resolving editing conflicts automatically. For example, an editing conflict may occur if two users of a shared word processor add a character simultaneously at the very same document position or assign different fonts concurrently to the very same paragraph.

2.1 Concurrency Control Libraries

Numerous concurrency control libraries offer the synchronization of various document copies in real-time as well as the reconciliation of emerging editing conflicts. The predominant concurrency control algorithm is the Operational Transformation (OT) algorithm which has been introduced by Ellis and Gibbs in 1989 [6]. In the meantime, the OT algorithm has been evolved to suit numerous document structures (e.g. SGML [5]) and to support advanced concurrency control operations (e.g. undo [9] or operation compression [13]).

ShareJS [7] is an example of a concurrency control library. The open-source ShareJS project is implemented in CoffeeScript which is compiled to JavaScript and thus, ShareJS can be embedded in arbitrary web applications. The OT-based library is capable of synchronizing simple string objects and JSON documents. However, the library does not allow syncing comprehensive data structures (e.g. graph models).

Apache Wave [2] emanated from the former Google Wave product and represents a full-fledged collaboration platform including an OT library. Its support for XML data structures allows implementing rich text editors that typically manage their data in some variation of a tree data structure. Adopting Apache Wave entails the usage of the Google Web Toolkit (GWT). Thereby, applications are implemented in Java and compiled to JavaScript. The rigid GWT development approach lacks interoperability with other development approaches.

SAP Gravity [10] represents another OT library which is part of SAP Process Flow, an SAP product for collaborative business process modeling. In terms of data structure support, SAP Gravity is more flexible than Apache Wave or ShareJS since it allows syncing graph models that may include cycles. Thus, it is capable to accommodate arbitrary business processes that can be expressed in BPMN. Since a tree is a connected graph without cycles, all tree data structures (e.g. XML) are also supported by SAP Gravity. The pure JavaScript library provided by SAP Gravity is suited for all standards-based web applications.

Even though OT libraries give developers a flexible means to implement collaboration capabilities at hand, adopting these solutions imposes various challenges. First, web developers have to become familiar with the Application Programming Interface (API). Second, using these APIs to introduce collaboration features into an existing application requires scattered and verbose source code changes. For example, developers have to capture local document changes and replay remote document changes. Third, some libraries (e.g. Apache Wave) are not interoperable with general-purpose frameworks or common development approaches. Considering these challenges we claim that an annotation-based approach can (1) reduce the entry barrier, (2) minimize required source code changes and (3) properly integrate with existing development approaches.

2.2 Transformation Approaches

Transformation approaches were pioneered by Sun et al. who advocated *transparent adaptation* [15] as a viable means to lower the development effort for groupware systems. The transparent adaption approach aims "to convert existing single-user applications into collaborative ones, without changing the source code of the original application" [15]. Thereby, a specific collaboration adapter links the application model to a generic collaboration engine in order to locally record and remotely replay manipulations. Sun et al. reported on the successful transformation of numerous prominent single-user tools such as Autodesk Maya [3], Microsoft Word [16] or Microsoft PowerPoint [15].

A second transformation approach targeting the conversion of single-user web applications was introduced in [8]. The conversion leverages a Generic Collaboration Infrastructure (GCI) that allows capturing and replaying Document Object Model (DOM) manipulations in an application-agnostic manner. The GCI transformation was adopted to successfully convert two widely-adopted, open-source editors (SVG-edit and CKEditor). In contrast to transparent adaptation requiring an application-specific collaboration adapter, the GCI represents a more efficient transformation technique depending solely on a tailored configuration file.

Both approaches incur limitations that we aim to address with the annotation-based ColADA solution. On the one hand, transparent adaption promotes the use of an extra collaboration adapter for each application which significantly increases the transformation effort. On the other hand, the GCI is only adoptable by web applications which expose a data model accommodated in the DOM. Web applications that expose an external data model represented by a specific JavaScript data structure are not supported.

3. THE COLADA APPROACH

We propose the ColADA approach to overcome the limitations of existing approaches, i.e. the boilerplate code required by concurrency control libraries, the substantial effort induced by transparent adaptation and the missing support of the GCI for external data models. Therefore, in this section, we introduce the design goals and challenges of the ColADA approach as well as framework requirements allowing to adopt ColADA. Moreover, we present the architecture of the ColADA system.

3.1 Design Goals

The overarching goal of our work is to increase programmer productivity in the context of the development of collaborative web applications. We refined this coarse-grained goal into the following objectives:

- Learnability: The approach should be easy to learn.
- Collaboration Functionality Completeness: The collaboration engine should expose mature and flexible concurrency control services.
- Interoperability: The collaborative applications should support cross-browser and cross-device scenarios, i.e. users can leverage shared editing capabilities using an arbitrary modern browser and an arbitrary Internet-ready device.
- Separation of Concerns: Collaboration features should be clearly separated from other functional aspects.

Framework Name	MVC Compliance	Notification Mechanism	Data Model Structure
AngularJS	Yes	Yes	Subgraph-based
Backbone.js	Yes	Yes	Scattered
Batman.js	Yes	Yes	Scattered
Cappuccino	Yes	Yes	Scattered
Ember.js	Yes	Yes	Scattered
JavaScript MVC	Yes	Yes	Scattered
Knockout	Yes	Yes	Subgraph-based
SAPUI5	Yes	Yes	Subgraph-based
SproutCore	Yes	Yes	Subgraph-based

Table 1: Catalog of MVC web frameworks

While learnability ensures that programmers can rapidly adopt the approach, collaboration functionality and interoperability assure that the envisioned collaboration engine meets the requirements of industrial-scale projects in terms of quality, flexibility as well as browser coverage. The separation of concerns goal facilitates product maintainability and eases the bundling of several product variants (i.e. single-user and multi-user versions).

To address the learnability objective, we adopt an annotation-based solution exhibiting a minimal set of annotations that enriches an existing framework instead of devising an extra framework. Hence, developers might leverage their existing skill set with respect to the host framework. To furthermore provide mature collaboration functionality, we aim to incorporate the operational transformation engine SAP Gravity that has a proven track record of serving real applications since it powers the industrial-strength product SAP Process Flow [10]. Interoperability is the most challenging objective since there is a myriad of browsers and a plentitude of Internet-ready devices whereas device characteristics (e.g. screen size) and browser characteristics (e.g. supported JavaScript libraries) strongly differ. Instead of dealing with varying browser or device characteristics, we plan to employ an existing web application framework providing a robust abstraction and hiding browser inconsistencies as well as device specifics. Satisfying the separation of concerns objective can also be accomplished through annotations since their declarative style establishes a distinctive isolation.

3.2 Framework Requirements

In the following, we derive from the design goals a set of requirements that a framework has to fulfill. First, a framework that qualifies for the enrichment with collaboration capabilities should enforce the *separation of the presentation and the data layer.* The availability of an encapsulated data model is the key to synchronize numerous application instances in a device- and browser-independent way since application models are a means to store data without including specifics about their presentation. Thus, the interoperability goal can be satisfied. Table 1 lists frameworks that enforce the established Model-View-Controller (MVC) structure [4].

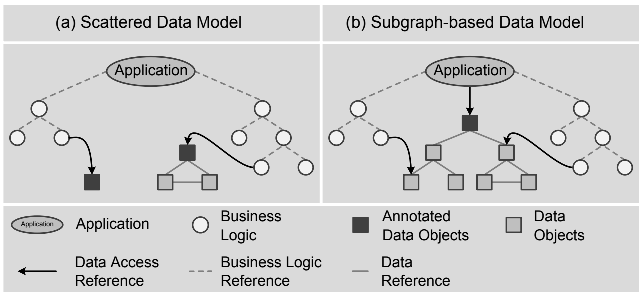


Figure 1: Classification of typical data model structures

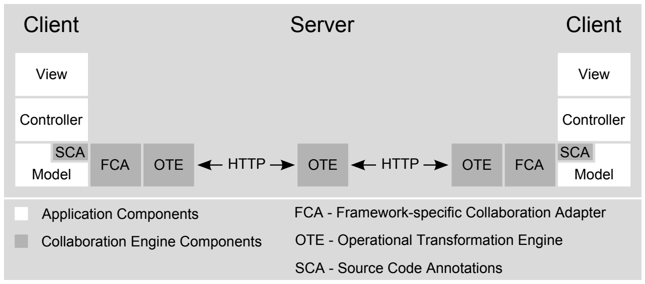

Client
Server
Client
View
Controller
SCA
Model
FCA
OTE
HTTP
OTE
HTTP
OTE
FCA
SCA
Model
View
Controller
Application Components
Collaboration Engine Components
FCA - Framework-specific Collaboration Adapter
OTE - Operational Transformation Engine
SCA - Source Code Annotations

Figure 2: Architecture of the ColADA system

A second framework requirement stems from the fact that we have to attach a collaboration engine providing suitable collaboration functionality (i.e. concurrency control services). Therefore, captured model changes have to be supplied to the collaboration engine which requires a *notification mechanism*. This notification mechanism allows the collaboration engine to react upon model changes. Hence, model manipulations can be recorded and propagated. As depicted in Table 1, all considered frameworks supply an appropriate mechanism to register event handlers.

A last requirement is induced by the adoption of an annotation-based approach supporting the learnability and separation of concerns objective. Annotations represent a viable means to declaratively mark the data model in order to configure the collaboration engine. Consequently, the *data model structure* determines the quantity of required annotations. Thus, we further analyzed the frameworks shown in Table 1 and established a data model classification depicted in Figure 1 grouping MVC applications into (1) scattered and (2) subgraph-based data model structures. Applications with a scattered data structure (cf. Figure 1a) expose numerous partial data models that are not interlinked. To discover and synchronize each model, the collaboration engine requires an annotation for each root node of a partial model. In contrast, applications with a subgraph-based data structure (cf. Figure 1b) require solely one annotation since the interlinked data structure can be completely discovered marking the single root node of the data model. Minimizing the number of source code annotations is essential to increase developer productivity and therefore, we only consider frameworks that enforce applications to expose a subgraph-based data model.

3.3 Architecture

After carving out framework requirements, we devised the ColADA architecture depicted in Figure 2. The distributed ColADA system consists of a server and an arbitrary number of clients. ColADA components are divided into white components belonging to the original application and grey components representing the collaboration engine. The white boxes illustrate a framework-based application including the model, the view and the controller. The controller mediates between view and model, i.e. once the user triggers view changes they are propagated to the model and vice versa.

Additionally, the application model represents the interface to the collaboration engine which captures local model manipulations and replays remote model modifications. The capture and replay logic is accommodated in the Framework-specific Collaboration Adapter (FCA). This FCA also includes the annotation processor that replaces introduced Source Code Annotations (SCAs) with JavaScript function calls once the application is loaded. Those inserted function calls are a means to register listeners as well as to attach replay handlers. To support proper document synchronization and conflict resolution, an Operational Transformation Engine (OTE) handles all sync mechanics. Thereby, the FCA supplies change notifications which are converted by the OTE into OT operations. Transforming concurrent OT operations allows to resolve conflicts and to maintain consistent document copies. For example, if two users simultaneously add a character at the first position of their document copy, the OTE adapts the indexes so that one character is added at the first position while the other character is inserted at the second position. Hence, the editing conflict is resolved and both document copies are consistent. Another responsibility of the OTE is to serialize OT operations in a JSON representation. Serialized OT operations are sent to a central server using common bi-directional, HTTP-based communication techniques such as long polling or HTTP streaming [17]. The server instance forwards the messages to all clients except the sender client. Once the message is delivered to a client, the JSON message is deserialized into an OT operation. In order to reconcile potential conflicts, this OT operation has to be transformed against concurrent local operations. Transformed OT operations are translated into model manipulations to sync the respective model instance.

4. COLADA SYSTEM IMPLEMENTATION

The ColADA architecture is materialized by a concrete implementation that we will discuss in the following. In essence, the implementation section focuses on the annotation language, the annotation incorporation workflow, the annotation replacement process as well as the sync procedures.

When selecting a specific framework for the ColADA implementation, we took into account the three identified framework requirements: (1) the availability of a notification mechanism, (2) the subgraph-based data structure and (3) the MVC compliance. Consequently, only four frameworks in Table 1 are eligible for adding shared editing capabilities. Due to the overwhelming adoption of the Knockout framework in recent months (i.e. approximately 1 million downloads in 2012), we selected Knockout [11] for the integration of an application-agnostic collaboration engine. Hence, we translated the generic ColADA architecture depicted in

Figure 2 into a concrete Knockout-specific implementation[1] whereas the framework-specific adapter is materialized by the Knockout Collaboration Adapter (KCA).

4.1 Annotation Language

We devised a Knockout-specific annotation language comprising the source code annotations @Sync and @Class. This compact annotation language is a means to configure sync processes in the following way:

- @Sync: The @Sync(modelName) annotation marks the Knockout model that should be synchronized among all application instances sharing the same session. The parameter *modelName* identifies the name of the JavaScript variable pointing to the data model.
- @Class: The @Class(className) annotation acts as a selector for object constructors. In order to allow for a proper replay of a local object creation at all remote sites, an object constructor has to be leveraged since the object creation might incur side effects. For example, creating a new object might entail to increment a global counter. This side effect of incrementing a counter cannot be replayed in a generic fashion and thus, the collaboration engine requires a handle to the actual object constructor.

4.2 Annotation Incorporation

To illustrate how source code annotations can be adopted to implement collaborative Knockout applications, we introduce the minimal example of a todo list application. This collaborative application should allow multiple users to concurrently add, remove or edit tasks that are organized in a list. Knockout applications commonly comprise two distinct parts: a view definition as well as a model definition which are automatically associated at runtime. To enhance such an application with collaboration support, the following steps are required:

- Annotation Insertion: Insert source code annotations in all files encapsulating data model definitions.
- Configuration: Complete a dedicated configuration by listing all files which contain annotations.
- KCA Import: Adapt the view definition in order to replace the original model import with the KCA import.

Figure 3 and Figure 4 show the collaboration-enabled view as well as the model for to the exemplary todo list application. The view definition (cf. Figure 3) mainly comprises regular HTML tags intermingled with a Knockout-specific *data-bind* attribute. While HTML tags define the UI to enter new tasks and to enumerate them in a dedicated list, the data-bind attribute establishes the link to the data model. The only difference between the original and the collaboration-enabled view definition is the script import section. Instead of embedding the original Knockout model (encapsulated in the `<!-- / -->` tags), the collaboration adapter *kca.js* has to be included. The kca.js script exploits a dedicated configuration file to retrieve associated model definitions. Eventually, the parser encapsulated in the kca.js locates all annotations and replaces them with the synchronization logic.

[1]Note that we also adopted the ColADA approach to implement a second collaboration adapter targeting the SAPUI5 framework [12] which is solely exploited for SAP-internal use cases.

```
...
  <!-- <script type="text/javascript" src="model.js"/> -->
  <script type="text/javascript" src="kca.js"/>
...

  <input data-bind="value: input"/>
  <button data-bind="click: addTask">Add Task</button>
  <ul data-bind="foreach: tasks">
     <li>
         <span data-bind="text: name"></span>
         <a href='#' data-bind="click: delete">Delete Task</a>
     </li>
  </ul>
...
```

Figure 3: Exemplary Knockout view including the *kca.js* script

```
// @Class("Task")
var Task = function (data) { this.name = ko.observable(data.name) }

Task.prototype.delete = function() { model.tasks.remove(this) }

// @Sync("model")
var model = { input: ko.observable(),
             tasks: ko.observableArray() }

model.addTask = function() {
       model.tasks.push(new Task({'name': model.input()}));
       model.input("") }

ko.applyBindings(model);
...
```

Figure 4: Exemplary Knockout model enhanced with annotations

In Figure 4, an annotated model definition associated to the view definition in Figure 3 is depicted. The @Class annotation marks the object constructor to allow for the creation of new task objects and @Sync points to the *model* variable to get a handle to the actual data model. Note that all annotations are encapsulated in JavaScript comments since JavaScript does not offer a native annotation concept. To illustrate the explained example, we additionally produced a screencast demonstrating the creation of the collaborative todo application and made it available on our ColADA page `http://vsr.informatik.tu-chemnitz.de/demo/ColADA/`.

4.3 Annotation Processing

To grasp the inner workings of the Knockout-specific implementation, it is crucial to understand the annotation processor that replaces annotations with JavaScript source code at runtime. The annotation processing starts by parsing all model definition files specified in the configuration and identifies inserted annotations. Those annotations are expanded to blocks of JavaScript code which for the @Class annotation is straightforward. The logic replacing the @Class annotation expands to a function call storing a reference to the constructor method in a global map. In contrast to this minimal substitution, the replacement of @Sync is challenging since the injected code has to bridge the gap between the Knockout model and the OTE which essentially enables the propagation of local manipulations and the replay of remote manipulations. Figure 5 depicts a skeleton of the inlined

```
     function traverseModel(knockoutModel) {
 I       ... // list includes all nodes of the Knockout model
         return koNodeList;
     }
     foreach(koNode in traverseModel(knockoutModel)) {
 II      koNode.setUUID();
         if(koNode.isType(Primitive)) {
 III         // create Gravity node and set inital value
             gravityNode = gravityModel.addNode(...);
 IV          // propagate local changes
             koNode.subscribe(function(newValue) {...});
 V           // subscribe to Gravity model changes
             gravityModel.addModelListener(...);
         }
         if(koNode.isType(Array)) {...}
     }
```

Figure 5: Skeleton of the JavaScript function replacing the @Sync annotation

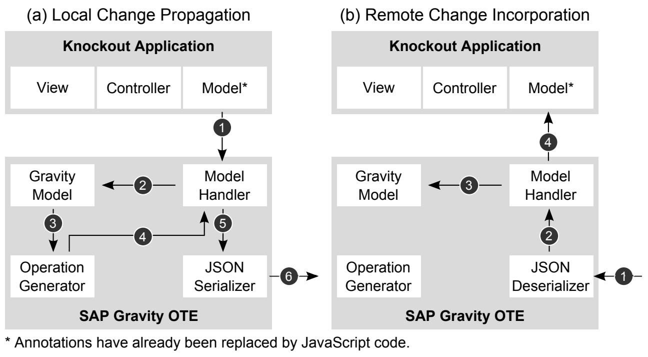


Figure 6: Synchronization workflows

function replacing the @Sync annotation. Note that this pseudocode is bound to the specific SAP Gravity OTE [10].

The SAP Gravity OTE consists of a graph model accommodating nodes, attributes and edges. A graph model is created using a dedicated JavaScript API that offers functions like createModel(), addNode(), etc. SAP Gravity synchronizes this specific graph model automatically adopting an OT-based conflict resolution scheme. Hence, to sync an application, the Knockout model has to be mapped to the Gravity data structure and vice versa. This bi-directional mapping is materialized by the functions depicted in Figure 5. Establishing the mapping is subdivided in (I) traversing the Knockout model, (II) assigning a unique ID to Knockout nodes, (III) creating Gravity counterparts for Knockout nodes, (IV) registering listeners on Knockout nodes to inform about local changes and (V) attaching listeners to Gravity nodes to replay remote changes. In comparison to inserting a one-line annotation, the complex inlined function supporting arbitrary Knockout models adds up to more than a thousand lines of JavaScript code. This complexity originates from the generic applicability of the function that supports the traversal of all graph-structured Knockout models, the mapping of various Knockout node types, the callback registration for different model change operations, etc. Note that adding the code for a specific Knockout model would drastically reduce the code complexity but the five major code blocks (cf. Figure 5) are still required.

4.4 Synchronization Workflows

After all annotations were replaced with corresponding JavaScript functions, the synchronization workflows as depicted in Figure 6 are executed by the browser's JavaScript engine. The synchronization is divided into two processes: the local change propagation (cf. Figure 6a) and the remote change incorporation (cf. Figure 6b).

The local change propagation encompasses various steps. First, listeners registered on the Knockout model translate all kinds of model manipulations (e.g. change, create or delete operations) into Gravity API calls and inform the model handler. As soon as the model handler is notified, the Gravity API calls are applied on the Gravity model. These changes to the Gravity model are observed by the operation generator which is in charge of extracting and grouping the resulting OT operations. Grouping OT operations is a specific Gravity OTE concept allowing to encapsulate numerous primitive OT operations in one complex OT operation that is executed in a transactional manner, i.e. complex operations are either completely executed or completely rolled back. For example, inserting a table in a word processor might comprise the creation of various table cells whereas this compound create-table operation can be easily translated to Gravity's complex operation concept. Aggregated OT operations are forwarded by the model handler to the JSON serializer. Eventually, the JSON serializer converts OT operation objects into a JSON representation that is transmitted to the server.

The server distributes the JSON messages to all clients except the sender client. Clients receiving JSON change sets trigger the remote change incorporation process (cf. Figure 6b). Initially, the JSON deserializer transforms JSON messages into JavaScript objects accommodating OT operations. The model handler then transforms these operations against concurrent local operations and the resulting transformed operations are applied to the Gravity model. Thereby, the operation generator is detached from the Gravity model to avoid propagating the change back to remote clients. Moreover, the model handler leverages the inlined code to reflect the changes in the Knockout model.

4.5 Application Deployment

Annotated web applications can be deployed on regular HTTP servers whereas an additional sync server is required. Once the application is deployed, a modern browser is sufficient to execute the annotation processing and the synchronization on the client side. Nevertheless, deploying a minified version requires the running of the annotation processor before the minification step since the minification removes the comments from the source code including annotations. Besides combining the annotation processor with a minification step, it is also feasible to produce a single-user product variant. Thereby, the minification step is run exclusively without the annotation processing step.

5. EVALUATION

To assess developer productivity, the Knockout-specific ColADA system and the set of annotations were leveraged in a developer study and compared to a traditional concurrency control library. In this section, we report on the selected evaluation characteristics, the adopted evaluation

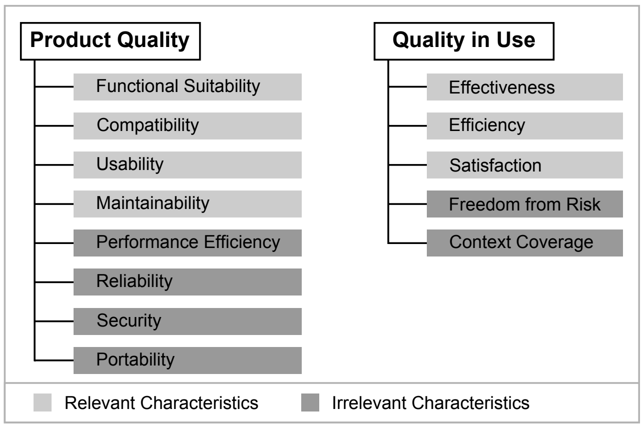


Figure 7: Product quality and quality in use model defined in the ISO/IEC 25010

procedure, the devised development task as well as the detected evaluation results.

5.1 Evaluation Characteristics

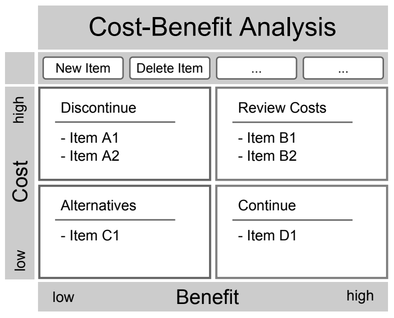
In order to yield meaningful results conducting a developer study, we embraced a number of established software metrics that target the evaluation of software systems. In particular, we took into account the product quality model and the quality in use model which are both defined in the ISO/IEC 25010 standard [1]. While the product quality model determines the static quality of a software system, the quality in use model emphasizes characteristics that are relevant in concrete usage scenarios. Both models encompass various quality aspects whereas some are not appropriate for the assessment of developer productivity. Figure 7 shows relevant characteristics that we embraced in the developer study as well as irrelevant characteristics that we chose to neglect.

5.2 Evaluation Procedure

Before conducting the actual developer study, several aspects had to be planned in advance. For example, developers were recruited, introductory lectures about the ColADA approach and the selected concurrency control library were prepared, a suitable task description for the development of a collaborative application was devised and a questionnaire to assess the selected quality characteristics was authored.

To limit the evaluation costs, we offered a three months course at the Dresden University of Technology instead of recruiting professional developers. Students were only eligible for the course if they were enrolled at the faculty of computer science. Eventually, eight students participated and completed a questionnaire assessing their programming expertise at the beginning of the course. The completed questionnaires showed that all students were familiar with numerous programming languages (e.g. Java, C, etc.). However, no student was acquainted with the development of shared editing applications.

The offered course was divided into three development sprints where students were asked to first develop a single-user application using the Knockout framework. Moreover, students had to enrich this single-user application with collaboration capabilities using, on the one hand, the devised



Functional Requirements:

(1) Cost and benefit items shall be classified in a 2 x 2 matrix.
(2) Matrix cells shall accommodate user-created items.
(3) Users shall be able to add and remove items.
(4) Users shall be able to move and reorder items using drag-and-drop.
(5) Matrix cells and the matrix itself shall expose a heading and the matrix axes shall exhibit a label.
(6) Headings, labels and items shall be editable.
(7) All user changes shall be synchronized in real-time and conflicts shall be automatically resolved.

Figure 8: Mockup and requirements of the cost-benefit analysis application

ColADA solution; and on the other hand, leveraging the concurrency control library SAP Gravity. Each development sprint started with a tailored lecture targeting (1) the Knockout framework, (2) the annotation-based programming model and (3) the SAP Gravity API.

The collaborative web application that was going to be developed to compare the traditional approach leveraging the Gravity API with the proposed annotation-based solution should support cost-benefit analyses. A Cost-Benefit Analysis (CBA) is a systematic process to justify an investment or to compare various projects by listing all positive and all negative factors. For example, a CBA might be used to review the construction of a new highway or to rethink the introduction of an enterprise resource planning system.

The development specification in Figure 8 defines the functional requirements and a mockup of the CBA application that students had to implement. After receiving an introductory session about programming Knockout applications, the task specification (cf. Figure 8) was distributed among all participants. In the first development sprint, students had to program the single-user application which served as the base application for the development of the collaborative CBA applications. In the second sprint, students programmed the collaborative CBA application adopting the Knockout collaboration adapter and in the third sprint, they were asked to introduce shared editing capabilities using the SAP Gravity API. Adopting the Gravity JavaScript API means students have to manually write the source code to sync the Knockout model and the Gravity model which includes traversing the models, registering callback functions, etc. (cf. Figure 5). During the entire development period, students had to work and implement their prototypes autonomously. However, a weekly meeting was setup to discuss issues with other participants or with a dedicated supervisor.

To properly analyze the students' work, various qualitative and quantitative data sources were captured. In particular, the following data sources were used to compare the two approaches for collaborative application development:

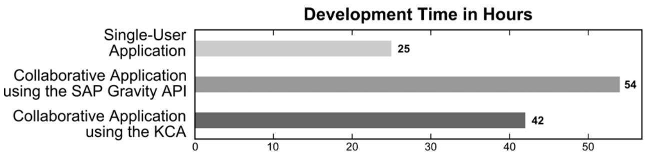


Figure 9: Calculated mean μ in the development time analysis

- Development Documentation: In every development sprint, students were asked to complete a form. This form was divided into two parts: the time recording and the issues section. In the time recordings' part, students had to enter a subtask description associated to the time spent for the completion. In the issues section, students explained encountered problems.
- Source Code: The source code handed in at the end of each sprint was analyzed to assess the fulfillment of the functional requirements and to measure the resulting lines of code. The lines-of-code analysis divided the code contributions into the individual programming languages (e.g. JavaScript, HTML, etc.).
- Questionnaire: After completing the three development sprints, all students had to fill out a questionnaire comprising 34 questions, 17 of which addressed the Gravity-based development and the other 17 aimed to assess the annotation-based development approach. The questionnaire depicted in Figure 11 was designed to evaluate product quality characteristics (cf. Q1 - Q12) as well as quality in use aspects (cf. Q13 - Q17).

5.3 Evaluation Results

To compare the effectiveness and efficiency of the ColADA approach with the conventional concurrency control library, we first employed two quantitative measures: (1) the development time and (2) the lines of code measure. Therefore, we exploited data collected in the form of development documentation and handed in source code. Only if all functional requirements were fulfilled, the collected data was included in the effectiveness and efficiency assessment. From eight students seven were able to completely finish the development of the single-user application as well as the implementation of the two collaborative applications.

Consequently, when calculating the mean μ of the total development time, we only considered the timesheets from seven students. On average, students spent 54 hours to get familiar with the Gravity API and to program the collaborative application in contrast to 42 hours adopting source code annotations (cf. Figure 9). Hence, employing the annotation-based approach could reduce the development effort by 22 percent. The overall development times of 54 hours and 42 hours respectively include 25 hours that were dedicated to the implementation of the single-user application. Thus, the actual development effort for introducing shared editing capabilities adds up to 29 hours versus 17 hours. This represents a 41 percent reduction when adopting the annotation-based approach. Even though the evaluation was only conducted with eight developers and solely included one specific concurrency control library (the Gravity API) as well as one specific annotation solution, the trend is apparent that configuring a collaboration engine using

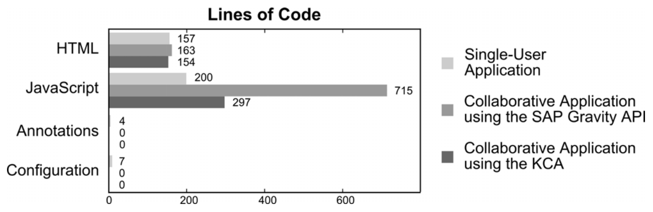


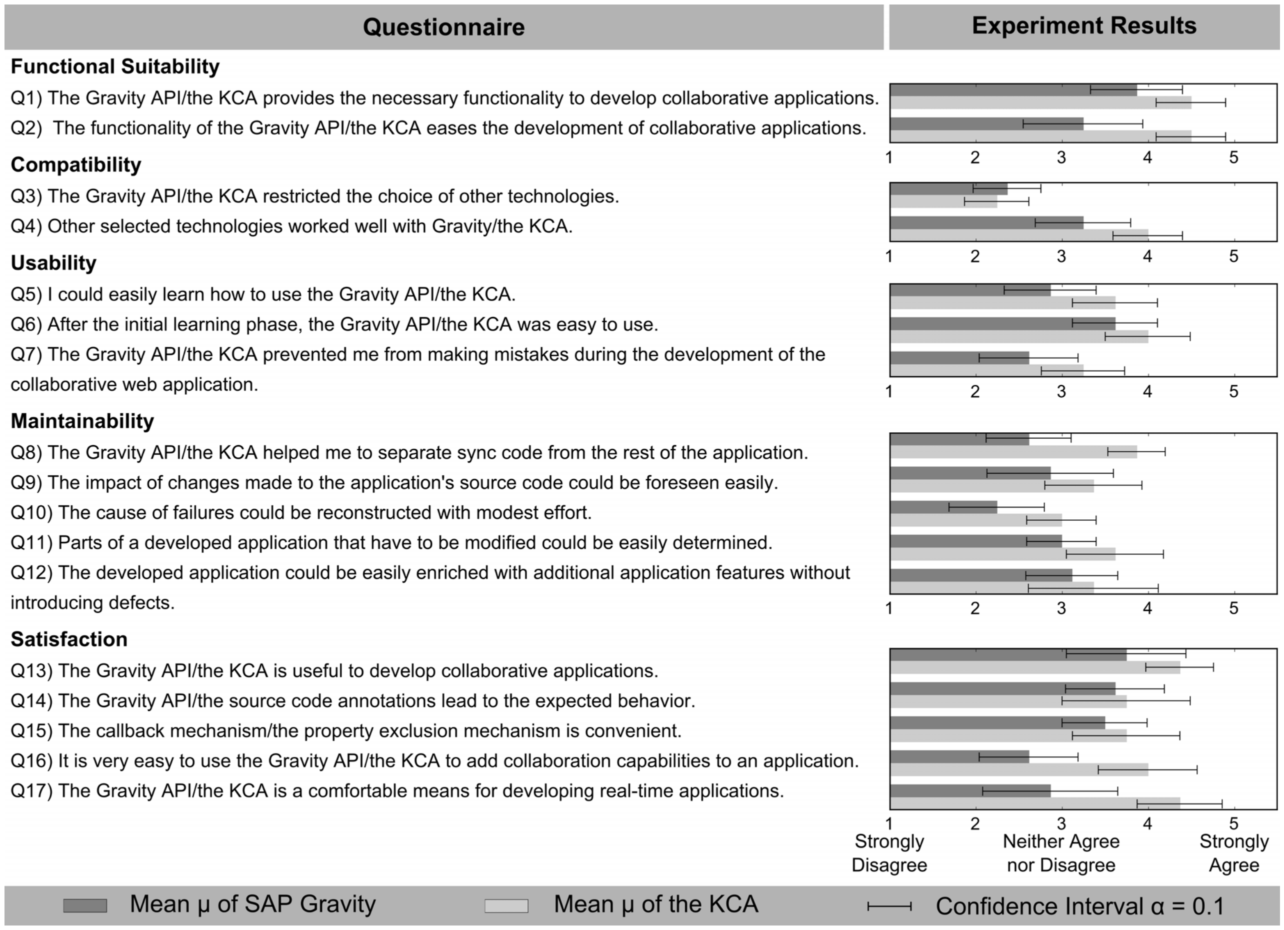
Figure 10: Calculated mean μ in the lines of code analysis

source code annotations is beneficial in terms of efficiency and can significantly outperform conventional collaboration libraries. In terms of effectiveness, both development approaches are suitable to develop collaborative applications since the resulting implementations were able to fulfill all functional requirements.

The second quantitative measure analyzed the Lines of Code (LoC) metric whereas seven valid source code contributions were included in the analysis. The code contributions were divided into the individual categories (1) HTML code, (2) JavaScript code, (3) annotation code and (4) configuration code. Figure 10 shows the LoC measurements whereas in each category the mean μ is depicted. One distinguishing factor between the use of the Gravity API and the use of annotations is the JavaScript LoC measure. On average, developers needed 97 lines of JavaScript code accompanied by 4 annotations and 7 configuration lines to inject collaboration capabilities in contrast to 515 lines of JavaScript code for adopting the Gravity API. This represents a considerable reduction of 81 percent in terms of JavaScript code when leveraging the proposed annotation-based approach. Even though the HTML LoC exposes only minor differences, the overall LoC measure resulting in 878 LoC versus 462 LoC once again shows a 47 percent source code reduction adopting the introduced ColADA solution. The substantial LoC reduction is another demonstration of the efficiency an annotation-based solution can deliver.

Besides employing quantitative measures, we also exploited qualitative evaluation techniques. Therefore, we created a questionnaire (cf. Figure 11) that targeted the selected evaluation characteristics (cf. Figure 7). For both development approaches, eight completed questionnaires were used to calculate the mean μ as well as the confidence interval [14]. While the mean μ is depicted in Figure 11 by grey bars, the confidence interval is visualized using black error bars. Confidence intervals are associated to a confidence level of 90 percent and the significance level of $\alpha = 0.1$. The confidence level expresses the likelihood that further equally conducted developer studies would also expose a mean within the limits of the confidence interval. Moreover, confidence intervals are a viable means to detect whether the difference of various mean constants (e.g. the means calculated for the two development approaches) are significant or not. If confidence intervals do not overlap, the differences of the means are significant [14]. If, on the contrary, confidence intervals do overlap, deriving an assured conclusion is not possible.

In general, the results of the experiment demonstrate that the ColADA approach constantly received superior ratings compared to the conventional approach leveraging the Gravity API. The functional suitability ratings could confirm that both approaches provide the necessary functionality



Questionnaire

Functional Suitability

Q1) The Gravity API/the KCA provides the necessary functionality to develop collaborative applications.

Q2) The functionality of the Gravity API/the KCA eases the development of collaborative applications.

Compatibility

Q3) The Gravity API/the KCA restricted the choice of other technologies.

Q4) Other selected technologies worked well with Gravity/the KCA.

Usability

Q5) I could easily learn how to use the Gravity API/the KCA.

Q6) After the initial learning phase, the Gravity API/the KCA was easy to use.

Q7) The Gravity API/the KCA prevented me from making mistakes during the development of the collaborative web application.

Maintainability

Q8) The Gravity API/the KCA helped me to separate sync code from the rest of the application.

Q9) The impact of changes made to the application's source code could be foreseen easily.

Q10) The cause of failures could be reconstructed with modest effort.

Q11) Parts of a developed application that have to be modified could be easily determined.

Q12) The developed application could be easily enriched with additional application features without introducing defects.

Satisfaction

Q13) The Gravity API/the KCA is useful to develop collaborative applications.

Q14) The Gravity API/the source code annotations lead to the expected behavior.

Q15) The callback mechanism/the property exclusion mechanism is convenient.

Q16) It is very easy to use the Gravity API/the KCA to add collaboration capabilities to an application.

Q17) The Gravity API/the KCA is a comfortable means for developing real-time applications.

Figure 11: Developer study questionnaire and the corresponding evaluation results

to develop collaborative applications (Q1: $\mu_{KCA} = 4.50$, $\mu_{Gra} = 3.88$). However, the ColADA approach could significantly outperform the Gravity approach with respect to the ease of development (Q2: $\mu_{KCA} = 4.50$, $\mu_{Gra} = 3.25$). Regarding the compatibility characteristics (Q3-Q4) developers stated that both programming methodologies did not restrict their choice of technology and that the technology interplay worked well. Even though the KCA ratings are slightly better (Q3 [2] : $\mu_{KCA} = 2.25$, $\mu_{Gra} = 2.38$, Q4: $\mu_{KCA} = 4.00$, $\mu_{Gra} = 3.25$), the difference is not substantial. The same trend with modestly better ratings for the KCA approach continued in the usability category where students were asked to assess the learnability, the ease of use and the error prevention. We would, however, have expected a larger difference, in particular in terms of learnability (Q5: $\mu_{KCA} = 3.63$, $\mu_{Gra} = 2.88$). Maintainability ratings once again exhibited considerable advantages for the KCA approach. In particular the separation of synchronization code from the rest of the application code is appropriately supported by the annotation-based approach (Q8: $\mu_{KCA} = 3.88$, $\mu_{Gra} = 2.63$). Nevertheless, the ability to detect and reconstruct failures leaves room for improvement. For the KCA approach error detection and debugging is especially cumbersome since annotations are replaced at runtime and consequently, the design time and the runtime definition differ. The satisfaction category once again assured that the KCA is easily adoptable (Q16: $\mu_{KCA} = 4.00$, $\mu_{Gra} = 2.63$) and represents a comfortable means to develop collaborative applications (Q17: $\mu_{KCA} = 4.38$, $\mu_{Gra} = 2.88$). The non-overlapping confidence intervals in Q16 and Q17 exhibit that this difference is significant.

[2]Note that in Q3 the lower the mean μ the better is the rating.

6. DISCUSSION

While exploring an annotation-based concurrency control solution, we observed numerous benefits and limitations. In the evaluation section, we reported on the benefits. For example, development time and lines of code were crucially reduced. In addition, evaluated software characteristics constantly received superior ratings compared to a traditional concurrency control API. In this section, limitations regarding the Knockout-based implementation are discussed.

Debugging Support: As described in Section 4.3 an annotation processor is in charge of replacing annotations with blocks of JavaScript code that are executed at runtime. When debugging the annotated application, developers are confronted with generated code and not with familiar annotations. This representation switch may hinder the debugging efficiency since developers have to adapt to the injected source code representing annotations. Nevertheless, the inlined JavaScript functions (cf. Figure 5) are fixed and thus, the replacement of an annotation is analog to stepping into the definition of a function call. The only difference is that the method body entered during debugging is associated to an annotation and not to a function call.

Notification Bypassing: The Knockout-specific notification mechanism is established through *observable* functions that allow inspecting model elements and thus, this mechanism is exploited by the KCA to capture model manipulations. Individual model properties (e.g. task name or task due date) can only be monitored if they are declared as Knockout observables. The Knockout framework offers three methods to declare observables: observable(), computed() and observableArray(). While the observable method allows declaring simple model properties (e.g. name), the computed method can declare aggregated properties (e.g. first and last name) and observableArray is used to declare arrays. However, if developers circumvent this Knockout-specific notification mechanism, the KCA has no means to record model manipulations and the sync mechanism breaks.

Runtime Model Enhancements: Once the kca.js script is loaded by the browser, the included annotation processor locates annotations and replaces them with JavaScript code that accommodates the listener and the replay logic. If the model definition changes at runtime (e.g. a task object is enhanced with a new *priority* property), the KCA will not take notice and will fail syncing this novel model property. Constantly examining all runtime objects for property enhancements could eliminate this limitation but at the cost of performance degradation. Since changing the model definition at runtime is rather exceptional, we did not adapt the current KCA implementation.

7. CONCLUSION

Incorporating shared editing capabilities in web applications using traditional concurrency control libraries is a time-consuming and tedious task. Therefore, we evaluated the ColADA solution which promised to increase development productivity since lightweight source code annotations are leveraged instead of using conventional collaboration libraries that induce the need to rigorously change the application's source code.

Through the transformation of the widely-adopted Knockout framework into a collaboration-enabled web application framework, we showed that source code annotations are in the first place a viable option to introduce collaboration features. Moreover, a developer study employing the adapted Knockout framework could affirm our hypothesis that the annotation-based ColADA approach can outperform a traditional concurrency control library. In the particular developer study, we compared the enriched Knockout framework with the SAP Gravity library and the results showed that the development time as well as the required source code changes can be substantially reduced when adopting an annotation-based solution. Additionally, the developer study exhibits that programmers are generally more satisfied with an annotation-based approach when comparing software quality characteristics like functional suitability, compatibility, usability, maintainability and satisfaction. Besides being beneficial for development efficiency, annotations are also a capable means to define multiple product variants (e.g. single-user and multi-user version) in one single code branch.

8. ACKNOWLEDGMENTS

This work was partially supported by funds from the European Commission (project OMELETTE, contract number 257635).

9. REFERENCES

[1] *ISO/IEC FDIS 25010 : 2010 (E) - Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models*. 2012.

[2] The Apache Software Foundation. Apache Wave. `http://incubator.apache.org/wave/`, 2012.

[3] Agustina, F. Liu, S. Xia, H. Shen, and C. Sun. CoMaya: Incorporating Advanced Collaboration Capabilities into 3D Digital Media Design Tools. In *Proc. CSCW*, pages 5–8, 2008.

[4] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*. John Wiley & Sons, 2007.

[5] A. H. Davis, C. Sun, and J. Lu. Generalizing Operational Transformation to the Standard General Markup Language. In *Proc. CSCW*, pages 58–67, 2002.

[6] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proc. SIGMOD*, pages 399–407. ACM, 1989.

[7] J. Gentle. ShareJS - Live Concurrent Editing in your App. `http://sharejs.org/`, 2012.

[8] M. Heinrich, F. Lehmann, T. Springer, and M. Gaedke. Exploiting Single-User Web Applications for Shared Editing - A Generic Transformation Approach. In *WWW*, pages 1057–1066, 2012.

[9] A. Prakash and M. J. Knister. A Framework for Undoing Actions in Collaborative Systems. *ACM Trans. Comput.-Hum. Interact.*, 1:295–330, 1994.

[10] A. Rickayzen. Collaborative Process Modeling. `http://scn.sap.com/community/bpm/business-process-modeling/blog/2012/03/20/`, 2012.

[11] S. Sanderson. Knockout : Home. `http://knockoutjs.com/`, 2012.

[12] UI Development Toolkit for HTML5 Developer Center. `http://scn.sap.com/community/developer-center/front-end`, 2012.

[13] H. Shen and C. Sun. Flexible Notification for Collaborative Systems. In *Proc. CSCW*, pages 77–86, 2002.

[14] G. Simpson, A. Roe, and R. Lewontin. *Quantitative Zoology: Revised Edition*. Dover Books on Biology, Psychology and Medicine. Dover Publications, 2003.

[15] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13:531–582, 2006.

[16] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging Single-user Applications for Multi-user Collaboration: the CoWord Approach. In *CSCW*, pages 162–171, 2004.

[17] N. C. Zakas. *Professional JavaScript for Web Developers*. Wrox, 2012.